Oracle® Solaris 11.4 Programming Interfaces Guide

Part No: E61059
August 2018

♦ ♦ ♦ CHAPTER 7

Socket Interfaces

This chapter describes the socket interface. Sample programs are included to illustrate key points. The following topics are discussed in this chapter:

- Socket creation, connection, and closure are discussed in "Socket Basics" on page 127.
- Client-Server architecture is discussed in "Client-Server Programs" on page 148.
- Advanced topics such as multicast and asynchronous sockets are discussed in "Advanced Socket Topics" on page 153.
- Interfaces used to implement the Stream Control Transmission Protocol (SCTP) are discussed in "Stream Control Transmission Protocol" on page 174.

Note - The interface that is described in this chapter is multithread safe. You can call applications that contain socket interface calls freely in a multithreaded application. The degree of concurrency that is available to applications is not specified.

Overview of Sockets

Sockets have been an integral part of SunOS releases since 1981. A socket is an endpoint of communication to which a name can be bound. A socket has a *type* and an associated process. Sockets were designed to implement the client-server model for interprocess communication where:

- The interface to network protocols needs to accommodate multiple communication protocols, such as TCP/IP, Xerox internet protocols (XNS), and the UNIX family.
- The interface to network protocols needs to accommodate server code that waits for connections and client code that initiates connections.
- Operations differ depending on whether communication is connection-oriented or connectionless.
- Application programs might want to specify the destination address of the datagrams that are being delivered instead of binding the address with the `open` call.

Sockets provide network protocols while behaving like UNIX files. Applications create sockets as needed. Sockets work with the `close`, `read`, `write`, `ioctl`, and `fcntl` interfaces. The operating system differentiates between the file descriptors for files and the file descriptors for sockets. For more information, see the `close(2)`, `read(2)`, `write(2)`, `ioctl(2)`, and `fcntl(2)` man pages.

Socket Libraries

The socket interface routines are in a library that must be linked with the application. The library `libsocket.so` is contained in `/usr/lib` with the rest of the system service libraries. Use `libsocket.so` for dynamic linking.

Socket Types

Socket types define the communication properties that are visible to a user. The Internet family sockets provide access to the TCP/IP transport protocols. The Internet family is identified by the value `AF_INET6`, for sockets that can communicate over both IPv6 and IPv4. The value `AF_INET` is also supported for source compatibility with old applications and for raw access to IPv4.

Oracle Solaris environment supports four types of sockets:

- *Stream* sockets enable processes to communicate using TCP. A stream socket provides a bidirectional, reliable, sequenced, and unduplicated flow of data with no record boundaries. After the connection has been established, data can be read from and written to these sockets as a byte stream. The socket type is `SOCK_STREAM`.
- *Datagram* sockets enable processes to use UDP to communicate. A datagram socket supports a bidirectional flow of messages. A process on a datagram socket might receive messages in a different order from the sending sequence. A process on a datagram socket might receive duplicate messages. Messages that are sent over a datagram socket might be dropped. Record boundaries in the data are preserved. The socket type is `SOCK_DGRAM`.
- *Raw* sockets provide access to ICMP. Raw sockets also provide access to other protocols based on IP that are not directly supported by the networking stack. These sockets are normally datagram oriented, although their exact characteristics are dependent on the interface provided by the protocol. Raw sockets are not for most applications. Raw sockets are provided to support the development of new communication protocols, or for access to more esoteric facilities of an existing protocol. Only superuser processes can use raw sockets. The socket type is `SOCK_RAW`.
- *SEQ* sockets support 1-to-N Stream Control Transmission Protocol (SCTP) connections. For more information, see "Stream Control Transmission Protocol" on page 174.

See “Selecting Specific Protocols” on page 158 for further information.

Interface Sets

Two sets of socket interfaces are available in Oracle Solaris. The BSD socket interfaces are provided and since SunOS version 5.7, the XNS 5 (UNIX03) socket interfaces are also provided. The XNS 5 interfaces differ from the BSD interfaces.

The XNS 5 socket interfaces are documented in the following man pages:

- `accept`(3C)
- `bind`(3C)
- `connect`(3C)
- `endhostent`(3C)
- `endnetent`(3C)
- `endprotoent`(3C)
- `endservent`(3C)
- `gethostbyaddr`(3C)
- `gethostbyname`(3C)
- `gethostent`(3C)
- `gethostname`(3C)
- `getnetbyaddr`(3C)
- `getnetbyname`(3C)
- `getnetent`(3C)
- `getpeername`(3C)
- `getprotobyname`(3C)
- `getprotobynumber`(3C)
- `getprotoent`(3C)
- `getservbyname`(3C)
- `getservbyport`(3C)
- `getservent`(3C)
- `getsockname`(3C)
- `getsockopt`(3C)
- `htonl`(3C)
- `htons`(3C)

- inet_addr(3C)
- inet_lnaof(3C)
- inet_makeaddr(3C)
- inet_netof(3C)
- inet_network(3C)
- inet_ntoa(3C)
- listen(3C)
- ntohl(3C)
- ntohs(3C)
- recv(3C)
- recvfrom(3C)
- recvmsg(3C)
- send(3C)
- sendmsg(3C)
- sendto(3C)
- sethostent(3C)
- setnetent(3C)
- setprotoent(3C)
- setservent(3C)
- setsockopt(3C)
- shutdown(3C)
- socket(3C)
- socketpair(3C)

The BSD Socket behavior is documented in the corresponding 3N man pages. In addition, the following interfaces have been added to section 3N:

- freeaddrinfo(3C)
- freehostent(3C)
- getaddrinfo(3C)
- getipnodebyaddr(3C)
- getipnodebyname(3C)
- getnameinfo(3C)
- inet_ntop(3C)
- inet_pton(3C)

See the standards(7) man page for information on building applications that use the XNS 5 (UNIX03) socket interface.

Socket Basics

This section describes the use of the basic socket interfaces.

Socket Creation

The `socket` call creates a socket in the specified family and of the specified type.

```
s = socket(family, type, protocol);
```

If the protocol is unspecified, the system selects a protocol that supports the requested socket type. The socket handle is returned. The socket handle is a file descriptor.

The *family* is specified by one of the constants that are defined in `sys/socket.h`. Constants that are named `AF_`*suite* specify the address format to use in interpreting names:

`AF_APPLETALK`	Apple Computer Inc. Appletalk network
`AF_INET6`	Internet family for IPv6 and IPv4
`AF_INET`	Internet family for IPv4 only
`AF_PUP`	Xerox Corporation PUP internet
`AF_UNIX`	UNIX file system

Socket types are defined in `sys/socket.h`. These types, `SOCK_STREAM`, `SOCK_DGRAM`, or `SOCK_RAW`, are supported by `AF_INET6`, `AF_INET`, and `AF_UNIX`. For more information, see the socket(3C) man page.

The following example creates a stream socket in the Internet family:

```
s = socket(AF_INET6, SOCK_STREAM, 0);
```

This call results in a stream socket. The TCP protocol provides the underlying communication. Set the *protocol* argument to `0`, the default, in most situations. You can specify a protocol other than the default, as described in "Advanced Socket Topics" on page 153.

Binding Local Names

A socket is created without a name. A remote process cannot refer to a socket until an address is bound to the socket. Processes that communicate are connected through addresses. In the Internet family, a connection is composed of local and remote addresses and local and remote ports. Duplicate ordered sets, such as: `protocol`, `local address`, `local port`, `foreign address`, `foreign port` cannot exist. In most families, connections must be unique.

The `bind` interface enables a process to specify the local address of the socket. This interface forms the `local address`, `local port` set. `connect` and `accept` complete a socket's association by fixing the remote half of the address tuple. For more information, see the `bind`(3C), `connect`(3C), and `accept`(3C) man pages.

The `bind` call is used as follows:

```
bind (s, name, namelen);
```

The socket handle is *s*. The bound name is a byte string that is interpreted by the supporting protocols. Internet family names contain an Internet address and port number.

This example demonstrates binding an Internet address.

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct sockaddr_in6 sin6;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
bzero (&sin6, sizeof (sin6));
sin6.sin6_family = AF_INET6;
sin6.sin6_addr.s6_addr = in6addr_arg;
sin6.sin6_port = htons(MYPORT);
bind(s, (struct sockaddr *) &sin6, sizeof sin6);
```

The content of the address `sin6` is described in “Address Binding” on page 159, where Internet address bindings are discussed.

Connection Establishment

Connection establishment is asymmetric, with one process acting as the client and the other as the server. The server binds a socket to a well-known address associated with the service and blocks on its socket for a connect request. An unrelated process can then connect to the server.

The client request services from the server by initiating a connection to the server's socket. On the client side, the `connect` call initiates a connection. In the Internet family, this connection might appear as:

```
struct sockaddr_in6 server;
...
connect(s, (struct sockaddr *)&server, sizeof server);
```

If the client's socket is unbound at the time of the connect call, the system automatically selects and binds a name to the socket. For more information, see "Address Binding" on page 159. This automatic selection is the usual way to bind local addresses to a socket on the client side.

To receive a client's connection, a server must perform two steps after binding its socket. The first step is to indicate how many connection requests can be queued. The second step is to accept a connection.

```
struct sockaddr_in6 from;
...
listen(s, 5);                /* Allow queue of 5 connections */
fromlen = sizeof(from);
newsock = accept(s, (struct sockaddr *) &from, &fromlen);
```

The socket handle *s* is the socket bound to the address to which the connection request is sent. The second parameter of `listen` specifies the maximum number of outstanding connections that might be queued. The `from` structure is filled with the address of the client. A `NULL` pointer might be passed. *fromlen* is the length of the structure. For more information, see the `listen`(3C) man page.

The `accept` routine normally blocks processes. `accept` returns a new socket descriptor that is connected to the requesting client. The value of *fromlen* is changed to the actual size of the address. For more information, see the `accept`(3C) man page.

A server cannot indicate that the server accepts connections from only specific addresses. The server can check the `from` address returned by `accept` and close a connection with an unacceptable client. A server can accept connections on more than one socket, or avoid blocking on the `accept` call. These techniques are presented in "Advanced Socket Topics" on page 153.

Connection Errors

An error is returned if the connection is unsuccessful, but an address bound by the system remains. If the connection is successful, the socket is associated with the server and data transfer can begin.

The following table lists some of the more common errors returned when a connection attempt fails.

TABLE 8 Socket Connection Errors

Socket Errors	Error Description
ENOBUFS	Lack of memory available to support the call.
EPROTONOSUPPORT	Request for an unknown protocol.
EPROTOTYPE	Request for an unsupported type of socket.
ETIMEDOUT	No connection established in specified time. This error happens when the destination host is down or when problems in the network cause in lost transmissions.
ECONNREFUSED	The host refused service. This error happens when a server process is not present at the requested address.
ENETDOWN or EHOSTDOWN	These errors are caused by status information delivered by the underlying communication interface.
ENETUNREACH or EHOSTUNREACH	These operational errors can occur because no route to the network or host exists. These errors can also occur because of status information returned by intermediate gateways or switching nodes. The status information that is returned is not always sufficient to distinguish between a network that is down and a host that is down.

Data Transfer

This section describes the interfaces to send and receive data. You can send or receive a message with the `read` and `write` interfaces as follows:

```
write(s, buf, sizeof buf);
read(s,  buf, sizeof buf);
```

You can also use `send` and `recv`.

```
send(s, buf, sizeof buf, flags);
recv(s, buf, sizeof buf, flags);
```

`send` and `recv` are similar to `read` and `write`, but the `flags` argument is required. The `flags` argument, which is defined in `sys/socket.h`, can be specified as a nonzero value if one or more of the following is required:

MSG_OOB	Send and receive out-of-band data
MSG_PEEK	Look at data without reading

`MSG_DONTROUTE`	Send data without routing packets

Out-of-band data is specific to stream sockets. When `MSG_PEEK` is specified with a `recv` call, any data present is returned to the user, but treated as still unread. The next `read` or `recv` call on the socket returns the same data. The option to send data without routing packets applied to the outgoing packets is currently used only by the routing table management process.

For more information, see the `read`(2), `write`(2), `send`(3C), and `recv`(3C) man pages.

Closing Sockets

A `SOCK_STREAM` socket can be discarded by a `close` interface call. If data is queued to a socket that delivers after a `close`, the protocol continues to transfer the data. The data is discarded if it remains undelivered after an arbitrary period. For more information, see the `close`(2) man page.

A `shutdown` closes `SOCK_STREAM` sockets. Both processes can acknowledge that they are no longer sending. This call has the form:

```
shutdown(s, how);
```

where `how` is defined as:

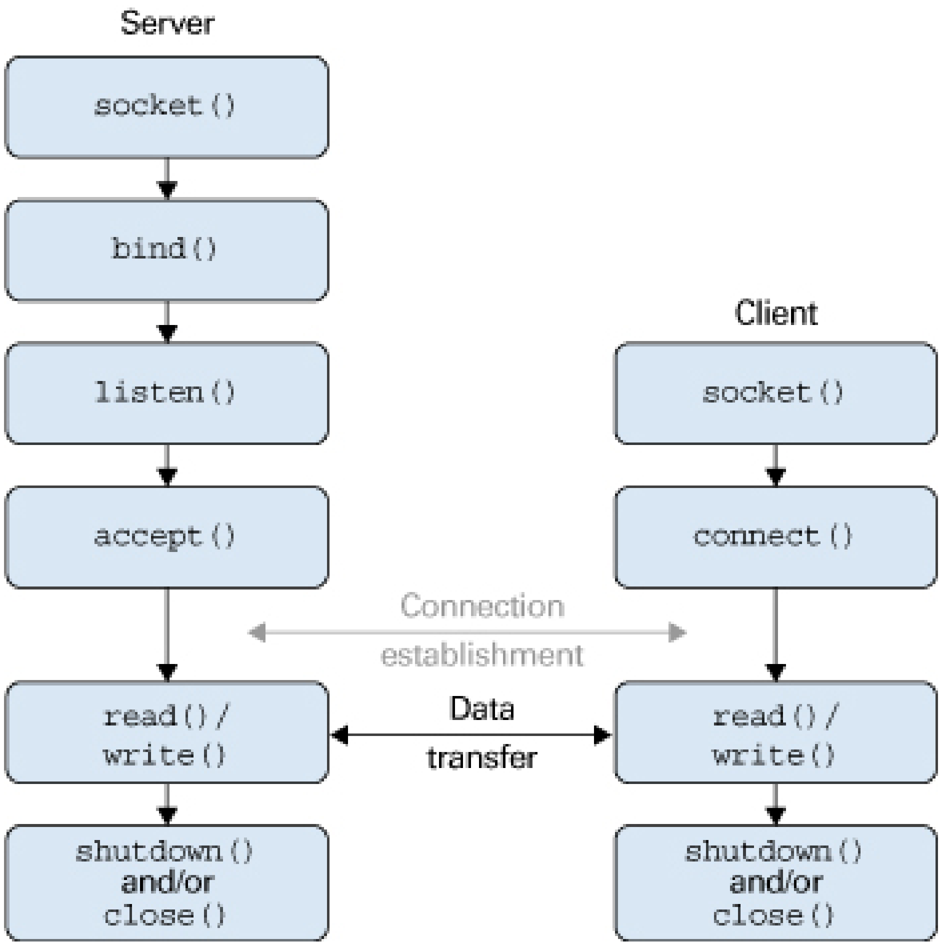
0	Disallows further data reception
1	Disallows further data transmission
2	Disallows further transmission and further reception

For more information on closing `SOCK_STREAM` sockets, see the `shutdown`(3C) man page.

Connecting Stream Sockets

The following two examples illustrate initiating and accepting an Internet family stream connection.

FIGURE 5 Connection-Oriented Communication Using Stream Sockets



The following example program is a server. The server creates a socket and binds a name to the socket, then displays the port number. The program calls `listen` to mark the socket as ready to accept connection requests and to initialize a queue for the requests. The rest of the program is an infinite loop. Each pass of the loop accepts a new connection and removes it from the queue, creating a new socket. The server reads and displays the messages from the socket and closes the socket. The use of `in6addr_any` is explained in "Address Binding" on page 159.

EXAMPLE 21 Accepting an Internet Stream Connection (Server)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
/*
 * This program creates a socket and then begins an infinite loop.
 * Each time through the loop it accepts a connection and prints
 * data from it. When the connection breaks, or the client closes
 * the connection, the program accepts a new connection.
*/
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    /* Create socket. */
    sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Bind socket using wildcards.*/
    bzero (&server, sizeof(server));
    server.sin6_family = AF_INET6;
    server.sin6_addr = in6addr_any;
    server.sin6_port = 0;
    if (bind(sock, (struct sockaddr *) &server, sizeof server)
            == -1) {
        perror("binding stream socket");
        exit(1);
    }
    /* Find out assigned port number and print it out. */
    length = sizeof server;
    if (getsockname(sock,(struct sockaddr *) &server, &length)
            == -1) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(server.sin6_port));
    /* Start accepting connections. */
    listen(sock, 5);
    do {
        msgsock = accept(sock,(struct sockaddr *) 0,(int *) 0);
        if (msgsock == -1)
            perror("accept");
        else do {
            memset(buf, 0, sizeof buf);
            if ((rval = read(msgsock,buf, sizeof(buf))) == -1)
                perror("reading stream message");
            if (rval == 0)
                printf("Ending connection\n");
            else
                /* assumes the data is printable */
                printf("-->%s\n", buf);
        } while (rval > 0);
        close(msgsock);
    } while(TRUE);
    /*
     * Since this program has an infinite loop, the socket "sock" is
     * never explicitly closed. However, all sockets are closed
     * automatically when a process is killed or terminates normally.
     */
    exit(0);
}
```

To initiate a connection, the client program in Example 22, “Internet Family Stream Connection (Client),” on page 134 creates a stream socket, then calls `connect`, specifying the address of the socket for connection. If the target socket exists, and the request is accepted, the connection is complete. The program can now send data. Data is delivered in sequence with no message boundaries. The connection is destroyed when either socket is closed. For more information about data representation routines such as `ntohl`, `ntohs`, `htons`, and `htonl`, see the `byteorder`(3C) man page.

EXAMPLE 22 Internet Family Stream Connection (Client)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "Half a league, half a league . . ."
/*
 * This program creates a socket and initiates a connection with
 * the socket given in the command line. Some data are sent over the
 * connection and then the socket is closed, ending the connection.
 * The form of the command line is: streamwrite hostname portnumber
 * Usage: pgm host port
 */
main(int argc, char *argv[])
{
    int sock, errnum, h_addr_index;
    struct sockaddr_in6 server;
    struct hostent *hp;
    char buf[1024];
    /* Create socket. */
    sock = socket( AF_INET6, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    /* Connect socket using name specified by command line. */
    bzero (&server, sizeof (server));
    server.sin6_family = AF_INET6;
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
/*
 * getipnodebyname returns a structure including the network address
 * of the specified host.
 */
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    h_addr_index = 0;
    while (hp->h_addr_list[h_addr_index] != NULL) {
        bcopy(hp->h_addr_list[h_addr_index], &server.sin6_addr,
                    hp->h_length);
        server.sin6_port = htons(atoi(argv[2]));
        if (connect(sock, (struct sockaddr *) &server,
                    sizeof (server)) == -1) {
            if (hp->h_addr_list[++h_addr_index] != NULL) {
                /* Try next address */
                continue;
            }
            perror("connecting stream socket");
            freehostent(hp);
            exit(1);
        }
        break;
    }
    freehostent(hp);
    if (write( sock, DATA, sizeof DATA) == -1)
        perror("writing on stream socket");
    close(sock);
    freehostent (hp);
    exit(0);
}
```

You can add support for one-to-one SCTP connections to stream sockets. The following example code adds the `-p` to an existing program, enabling the program to specify the protocol to use.

EXAMPLE 23 Adding SCTP Support to a Stream Socket

```
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <errno.h>

int
main(int argc, char *argv[])
{
    struct protoent *proto = NULL;
    int c;
    int s;
    int protocol;

    while ((c = getopt(argc, argv, "p:")) != -1) {
        switch (c) {
        case 'p':
            proto = getprotobyname(optarg);
            if (proto == NULL) {
                fprintf(stderr, "Unknown protocol: %s\n",
                        optarg);
                return (-1);
            }
            break;
        default:
            fprintf(stderr, "Unknown option: %c\n", c);
            return (-1);
        }
    }

    /* Use the default protocol, which is TCP, if not specified. */
    if (proto == NULL)
        protocol = 0;
    else
        protocol = proto->p_proto;

    /* Create a IPv6 SOCK_STREAM socket of the protocol. */
    if ((s = socket(AF_INET6, SOCK_STREAM, protocol)) == -1) {
        fprintf(stderr, "Cannot create SOCK_STREAM socket of type %s: "
                "%s\n", proto != NULL ? proto->p_name : "tcp",
                strerror(errno));
        return (-1);
    }
    printf("Success\n");
    return (0);
}
```

Input/Output Multiplexing

Requests can be multiplexed among multiple sockets or multiple files. Use `select` to multiplex:

```
#include <sys/time.h>
#include <sys/types.h>
#include <sys/select.h>
...
fd_set readmask, writemask, exceptmask;
struct timeval timeout;
...
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```

The first argument of `select` is the number of file descriptors in the lists pointed to by the next three arguments.

The second, third, and fourth arguments of `select` point to three sets of file descriptors: a set of descriptors to read on, a set to write on, and a set on which exception conditions are accepted. Out-of-band data is the only exceptional condition. You can designate any of these pointers as a properly cast null. Each set is a structure that contains an array of long integer bit masks. Set the size of the array with `FD_SETSIZE`, which is defined in `select.h`. The array is long enough to hold one bit for each `FD_SETSIZE` file descriptor.

The macros `FD_SET` (*fd*, &*mask*) and `FD_CLR` (*fd*, &*mask*) add and delete, respectively, the file descriptor *fd* in the set `mask`. The set should be zeroed before use and the macro `FD_ZERO` (*&mask*) clears the set `mask`.

The fifth argument of `select` enables the specification of a timeout value. If the `timeout` pointer is `NULL`, `select` blocks until a descriptor is selectable, or until a signal is received. If the fields in `timeout` are set to `0`, `select` polls and returns immediately.

The `select` routine returns the number of file descriptors that are selected, or a zero if the timeout has expired. The `select` routine returns `-1` for an error or interrupt, with the error number in *errno* and the file descriptor masks unchanged. For a successful return, the three sets indicate which file descriptors are ready to be read from, written to, or have exceptional conditions pending.

Test the status of a file descriptor in a select mask with the `FD_ISSET` (*fd*, &*mask*) macro. The macro returns a nonzero value if *fd* is in the set `mask`. Otherwise, the macro returns zero. Use select followed by a FD_ISSET (*fd*, *&mask*) macro on the read set to check for queued connect requests on a socket. For more information, see the select(3C) man page.

The following example shows how to select on a listening socket for readability to determine when a new connection can be picked up with a call to accept. The program accepts connection requests, reads data, and disconnects on a single socket.

EXAMPLE 24 Using select(3C) to Check for Pending Connections

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time/h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define TRUE 1
/*
 * This program uses select to check that someone is
 * trying to connect before calling accept.
 */
main() {
    int sock, length;
    struct sockaddr_in6 server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;
    /* Open a socket and bind it as in previous examples. */
    /* Start accepting connections. */
    listen(sock, 5);
    do {
        FD_ZERO(&ready);
        FD_SET(sock, &ready);
        to.tv_sec = 5;
        to.tv_usec = 0;
        if (select(sock + 1, &ready, (fd_set *)0,
                   (fd_set *)0, &to) == -1) {
            perror("select");
            continue;
        }
        if (FD_ISSET(sock, &ready)) {
            msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
            if (msgsock == -1)
                perror("accept");
            else do {
                memset(buf, 0, sizeof buf);
                if ((rval = read(msgsock, buf, sizeof(buf))) == -1)
                    perror("reading stream message");
                else if (rval == 0)
                    printf("Ending connection\n");
                else
                    printf("-->%s\n", buf);
            } while (rval > 0);
            close(msgsock);
        } else
            printf("Do something else\n");
        } while (TRUE);
    exit(0);
}
```

In previous versions of the `select` routine, its arguments were pointers to integers instead of pointers to `fd_sets`. This style of call still works if the number of file descriptors is smaller than the number of bits in an integer.

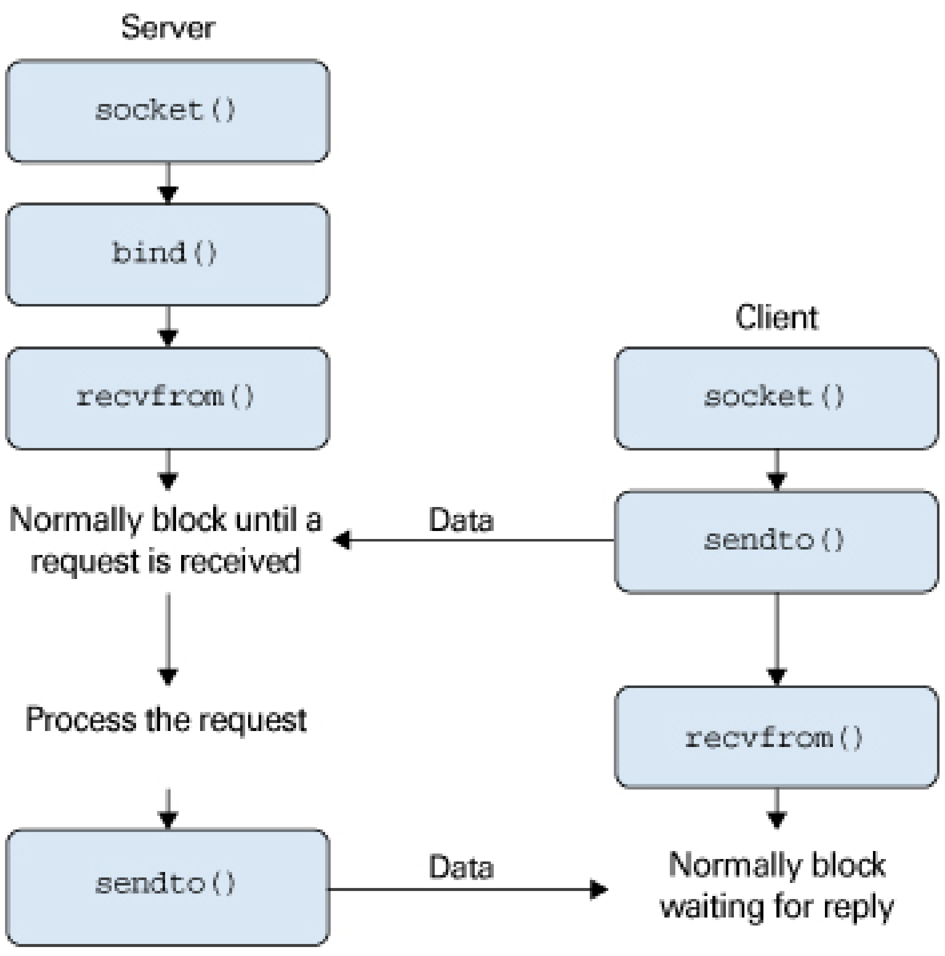
The `select` routine provides a synchronous multiplexing scheme. The `SIGIO` and `SIGURG` signals, which is described in "Advanced Socket Topics" on page 153, provide asynchronous notification of output completion, input availability, and exceptional conditions.

Datagram Sockets

A datagram socket provides a symmetric data exchange interface without requiring connection establishment. Each message carries the destination address. The following figure shows the flow of communication between server and client.

The `bind` step for the server is optional.

FIGURE 6 Connectionless Communication Using Datagram Sockets



Create datagram sockets as described in "Socket Creation" on page 127. If a particular local address is needed, the `bind` operation must precede the first data transmission. Otherwise, the system sets the local address or port when data is first sent. Use `sendto` to send data. For more information, see the `bind`(3C) and `sendto`(3C) man pages.

```
sendto(s, buf, buflen, flags, (struct sockaddr *) &to, tolen);
```

The *s*, *buf*, *buflen*, and *flags* parameters are the same as in connection-oriented sockets. The *to* and *tolen* values indicate the address of the intended recipient of the message. A locally detected error condition, such as an unreachable network, causes a return of `-1` and *errno* to be set to the error number.

```
recvfrom(s, buf, buflen, flags, (struct sockaddr *) &from, &fromlen);
```

To receive messages on a datagram socket, `recvfrom` is used. Before the call, *fromlen* is set to the size of the *from* buffer. On return, `fromlen` is set to the size of the address from which the datagram was received. For more information, see the `recvfrom`(3C) man page.

Datagram sockets can also use the `connect` call to associate a socket with a specific destination address. The socket can then use the `send` call. Any data that is sent on the socket that does not explicitly specify a destination address is addressed to the connected peer. Only the data that is received from that peer is delivered. A socket can have only one connected address at a time. A second `connect` call changes the destination address. Connect requests on datagram sockets return immediately. The system records the peer's address. Neither `accept` nor `listen` are used with datagram sockets. For more information, see the `send`(3C), `connect`(3C), `accept`(3C), and `listen`(3C) man pages.

A datagram socket can return errors from previous `send` calls asynchronously while the socket is connected. The socket can report these errors on subsequent socket operations. Alternately, the socket can use an option of `getsockopt`, `SO_ERROR` to interrogate the error status. For more information, see the `getsockopt`(3C) man page.

The following example code shows how to send an Internet call by creating a socket, binding a name to the socket, and sending the message to the socket.

EXAMPLE 25 Sending an Internet Family Datagram

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define DATA "The sea is calm, the tide is full . . ."
/*
 * Here I send a datagram to a receiver whose name I get from
 * the command line arguments. The form of the command line is:
 * dgramsend hostname portnumber
 */
main(int argc, char *argv[])
{
    int sock, errnum;
    struct sockaddr_in6 name;
    struct hostent *hp;
    /* Create socket on which to send. */
    sock = socket(AF_INET6,SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /*
     * Construct name, with no wildcards, of the socket to ``send''
     * to. getinodebyname returns a structure including the network
     * address of the specified host. The port number is taken from
     * the command line.
     */
    hp = getipnodebyname(argv[1], AF_INET6, AI_DEFAULT, &errnum);
    if (hp == (struct hostent *) 0) {
        fprintf(stderr, "%s: unknown host\n", argv[1]);
        exit(2);
    }
    bzero (&name, sizeof (name));
    memcpy((char *) &name.sin6_addr, (char *) hp->h_addr,
       hp->h_length);
    name.sin6_family = AF_INET6;
    name.sin6_port = htons(atoi(argv[2]));
    /* Send message. */
    if (sendto(sock,DATA, sizeof DATA ,0,
        (struct sockaddr *) &name,sizeof name) == -1)
        perror("sending datagram message");
    close(sock);
    exit(0);
}
```

The following sample code shows how to read an Internet call by creating a socket, binding a name to the socket, and then reading from the socket.

EXAMPLE 26 Reading Internet Family Datagrams

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
/*
 * This program creates a datagram socket, binds a name to it, then
 * reads from the socket.
 */
 main()
{
    int sock, length;
    struct sockaddr_in6 name;
    char buf[1024];
    /* Create socket from which to read. */
    sock = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name with wildcards. */
    bzero (&name, sizeof (name));
    name.sin6_family = AF_INET6;
    name.sin6_addr = in6addr_any;
    name.sin6_port = 0;
    if (bind (sock, (struct sockaddr *)&name, sizeof (name)) == -1) {
        perror("binding datagram socket");
        exit(1);
    }
    /* Find assigned port value and print it out. */
    length = sizeof(name);
    if (getsockname(sock,(struct sockaddr *) &name, &length)
            == -1) {
        perror("getting socket name");
        exit(1);
    }
    printf("Socket port #%d\n", ntohs(name.sin6_port));
    /* Read from the socket. */
    if (read(sock, buf, 1024) == -1 )
        perror("receiving datagram packet");
    /* Assumes the data is printable */
    printf("-->%s\n", buf);
    close(sock);
    exit(0);
}
```

Standard Routines

This section describes the routines that you can use to locate and construct network addresses. Unless otherwise stated, interfaces presented in this section apply only to the Internet family.

Locating a service on a remote host requires many levels of mapping before the client and server communicate. A service has a name for human use. The service and host names must translate to network addresses. Finally, the network address must be usable to locate and route to the host. The specifics of the mappings can vary between network architectures.

Standard routines map host names to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers. The standard routines also indicate the appropriate protocol to use in communicating with the server process. The file `netdb.h` must be included when using any of these routines.

Host and Service Names

The interfaces `getaddrinfo`, `getnameinfo`, `gai_strerror`, and `freeaddrinfo` provide a simplified way to translate between the names and addresses of a service on a host. These interfaces are more recent than the `getipnodebyname`, `gethostbyname`, and `getservbyname` APIs. Both IPv6 and IPv4 addresses are handled transparently. For more information, see the `getaddrinfo`(3C), `getnameinfo`(3C), `gai_strerror`(3C), `freeaddrinfo`(3C), `getipnodebyname`(3C), `gethostbyname`(3C), and `getservbyname`(3C) man pages.

The `getaddrinfo` routine returns the combined address and port number of the specified host and service names. Because the information returned by `getaddrinfo` is dynamically allocated, the information must be freed by `freeaddrinfo` to prevent memory leaks. `getnameinfo` returns the host and services names associated with a specified address and port number. Call `gai_strerror` to print error messages based on the `EAI_xxx` codes returned by `getaddrinfo` and `getnameinfo`.

An example of using `getaddrinfo` follows.

```
struct addrinfo        *res, *aip;
struct addrinfo        hints;
int                    error;

/* Get host address.  Any type of address will do. */
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
    (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
    gai_strerror(error), hostname, servicename);
    return (-1);
}
```

After processing the information returned by `getaddrinfo` in the structure pointed to by `res`, the storage should be released by `freeaddrinfo(res)`.

The `getnameinfo` routine is particularly useful in identifying the cause of an error, as in the following example:

```
struct sockaddr_storage faddr;
int                     sock, new_sock, sock_opt;
socklen_t               faddrlen;
int                     error;
char                    hname[NI_MAXHOST];
char                    sname[NI_MAXSERV];
...
faddrlen = sizeof (faddr);
new_sock = accept(sock, (struct sockaddr *)&faddr, &faddrlen);
if (new_sock == -1) {
    if (errno != EINTR && errno != ECONNABORTED) {
        perror("accept");
    }
    continue;
}
error = getnameinfo((struct sockaddr *)&faddr, faddrlen, hname,
    sizeof (hname), sname, sizeof (sname), 0);
if (error) {
    (void) fprintf(stderr, "getnameinfo: %s\n",
        gai_strerror(error));
} else {
    (void) printf("Connection from %s/%s\n", hname, sname);
}
```

Host Names – `hostent`

An Internet host-name-to-address mapping is represented by the `hostent` structure as defined in `gethostent`gethostent(3C):

```
struct hostent {
    char  *h_name;            /* official name of host */
    char  **h_aliases;        /* alias list */
    int   h_addrtype;         /* hostaddrtype(e.g.,AF_INET6) */
    int   h_length;           /* length of address */
    char  **h_addr_list;      /* list of addrs, null terminated */
};
/*1st addr, net byte order*/
#define h_addr h_addr_list[0]
```

`getipnodebyname`(3C)	Maps an Internet host name to a `hostent` structure
`getipnodebyaddr`(3C)	Maps an Internet host address to a `hostent` structure
`freehostent`(3C)	Frees the memory of a `hostent` structure

inet_ntop(3C)	Maps an Internet host address to a string

The routines return a hostent structure that contains the name of the host, its aliases, the address type, and a NULL-terminated list of variable length addresses. The list of addresses is required because a host can have many addresses. The h_addr definition is for backward compatibility, and is the first address in the list of addresses in the hostent structure.

Network Names – netent

The routines to map network names to numbers and the reverse return a netent structure:

```
/*
 * Assumes that a network number fits in 32 bits.
 */
struct netent {
   char     *n_name;      /* official name of net */
   char     **n_aliases;  /* alias list */
   int      n_addrtype;   /* net address type */
   int      n_net;        /* net number, host byte order */
};
```

getnetbyname(3C), getnetbyaddr_r(3C), and getnetent(3C) are the network counterparts to the host routines previously described.

Protocol Names – protoent

The protoent structure defines the protocol-name mapping used with getprotobyname, getprotobynumber, and getprotoent and are defined in getprotoent:

```
struct protoent {
    char    *p_name;       /* official protocol name */
    char    **p_aliases    /* alias list */
    int     p_proto;       /* protocol number */
};
```

For more information, see the getprotobyname(3C), getprotobynumber(3C), and getprotoent(3C) man pages.

Service Names – `servent`

An Internet family service resides at a specific, well-known port, and uses a particular protocol. A service-name-to-port-number mapping is described by the `servent` structure that is defined in `getprotoent`:

```
struct servent {
    char    *s_name;        /* official service name */
    char    **s_aliases;    /* alias list */
    int     s_port;         /* port number, network byte order */
    char    *s_proto;       /* protocol to use */
};
```

`getservbyname`(3C) maps service names and, optionally, a qualifying protocol to a `servent` structure. The following call returns the service specification of a telnet server that is using any protocol.

```
sp = getservbyname("telnet", (char *) 0);
```

The following call returns the telnet server that uses the TCP protocol.

```
sp = getservbyname("telnet", "tcp");
```

`getservbyport` and `getservent` are also provided. `getservbyport` has an interface that is similar to the interface used by `getservbyname`. You can specify an optional protocol name to qualify lookups For more information, see the `getservbyname`(3C), `getservbyport`(3C), and `getservent`(3C) man pages.

Other Routines

Several other routines that simplify manipulating names and addresses are available. The following table summarizes the routines for manipulating variable-length byte strings and byte-swapping network addresses and values.

TABLE 9 Runtime Library Routines

Interface	Synopsis
`memcmp`	Compares byte-strings; `0` if same, not `0` otherwise. For more information, see the `memcmp`(3C) man page.
`memcpy`	Copies *n* bytes from *s2* to *s1*. For more information, see the `memcpy`(3C) man page.

Interface	Synopsis
`memset`	Sets *n* bytes to `value` starting at `base`. For more information, see the `memset`(3C) man page.
`htonl`	32-bit quantity from host into network byte order. For more information, see the `htonl`(3C) man page.
`htons`	16-bit quantity from host into network byte order. For more information, see the `htons`(3C) man pages.
`ntohl`	32-bit quantity from network into host byte order. For more information, see the `ntohl`(3C) man page.
`ntohs`	16-bit quantity from network into host byte order. For more information, see the `ntohs`(3C) man page.

The byte-swapping routines are provided because the operating system expects addresses to be supplied in network order. On some architectures, the host byte ordering is different from network byte order, so programs must sometimes byte-swap values. Routines that return network addresses do so in network order. Byte-swapping problems occur only when interpreting network addresses. For example, the following code formats a TCP or UDP port:

```
printf("port number %d\n", ntohs(sp->s_port));
```

On systems that do not need these routines, the routines are defined as null macros.

Client-Server Programs

The most common form of distributed application is the client-server model. In this scheme, client processes request services from a server process.

An alternate scheme is a service server that can eliminate dormant server processes. An example is `inetd`, the Internet service daemon. `inetd` listens at a variety of ports, determined at startup by reading a configuration file. When a connection is requested on an `inetd` serviced port, `inetd` spawns the appropriate server to serve the client. Clients are unaware that an intermediary has played any part in the connection. For more information, see "`inetd` Daemon" on page 164.

Sockets and Servers

Most servers are accessed at well-known Internet port numbers or UNIX family names. The service `rlogin` is an example of a well-known UNIX family name. The main loop of a remote login server is shown in Example 27, "Server Main Loop," on page 149.

The server dissociates from the controlling terminal of its invoker unless the server is operating in DEBUG mode.

```
(void) close(0);
(void) close(1);
(void) close(2);
(void) open("/", O_RDONLY);
(void) dup2(0, 1);
(void) dup2(0, 2);
setsid();
```

Dissociating prevents the server from receiving signals from the process group of the controlling terminal. After a server has dissociated from the controlling terminal, the server cannot send reports of errors to the terminal. The dissociated server must log errors with syslog. For more information, see the syslog(3C) man page.

The server gets its service definition by calling getaddrinfo. For more information, see the getaddrinfo(3C) man page.

```
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;
error = getaddrinfo(NULL, "rlogin", &hints, &api);
```

The result, which is returned in api, contains the Internet port at which the program listens for service requests. Some standard port numbers are defined in /usr/include/netinet/in.h.

The server then creates a socket, and listens for service requests. The bind routine ensures that the server listens at the expected location. Because the remote login server listens at a restricted port number, the server runs as superuser. The main body of the server is the following loop.

EXAMPLE 27 Server Main Loop

```
/* Wait for a connection request. */
for (;;) {
    faddrlen = sizeof (faddr);
    new_sock = accept(sock, (struct sockaddr *)api->ai_addr,
            api->ai_addrlen)
    if (new_sock == -1) {
        if (errno != EINTR && errno != ECONNABORTED) {
            perror("rlogind: accept");
        }
        continue;
    }
    if (fork() == 0) {
        close (sock);
        doit (new_sock, &faddr);
    }
    close (new_sock);
}
/*NOTREACHED*/
```

`accept` blocks messages until a client requests service. `accept` also returns a failure indication if `accept` is interrupted by a signal, such as `SIGCHLD`. The return value from `accept` is checked, and an error is logged with `syslog`, if an error occurs. For more information, see the `accept`(3C) and `syslog`(3C) man pages.

The server then forks a child process, and invokes the main body of the remote login protocol processing. The socket used by the parent to queue connection requests is closed in the child. The socket created by `accept` is closed in the parent. The address of the client is passed to the server application's `doit()` routine, which authenticates the client.

Sockets and Clients

This section describes the steps taken by a client process. As in the server, the first step is to locate the service definition for a remote login.

```
bzero(&hints, sizeof (hints));
hints.ai_flags = AI_ALL|AI_ADDRCONFIG;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(hostname, servicename, &hints, &res);
if (error != 0) {
    (void) fprintf(stderr, "getaddrinfo: %s for host %s service %s\n",
        gai_strerror(error), hostname, servicename);
    return (-1);
}
```

`getaddrinfo` returns the head of a list of addresses in `res`. The desired address is found by creating a socket and trying to connect to each address returned in the list until one works.

```
for (aip = res; aip != NULL; aip = aip->ai_next) {
    /*
     * Open socket.  The address type depends on what
     * getaddrinfo() gave us.
     */
    sock = socket(aip->ai_family, aip->ai_socktype,
          aip->ai_protocol);
    if (sock == -1) {
        perror("socket");
        freeaddrinfo(res);
        return (-1);
    }

    /* Connect to the host. */
    if (connect(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
        perror("connect");
        (void) close(sock);
        sock = -1;
        continue;
    }
    break;
}
```

The socket has been created and has been connected to the desired service. The `connect` routine implicitly binds `sock`, because `sock` is unbound.

Connectionless Servers

Some services use datagram sockets. The `rwho` service provides status information on hosts that are connected to a local area network. Avoid running `in.rwhod` because `in.rwho` causes heavy network traffic. The `rwho` service broadcasts information to all hosts connected to a particular network. The `rwho` service is an example of datagram socket use.

A user on a host that is running the `rwho` server can get the current status of another host with `ruptime`. For more information, see the `rwho`(1), `in.rwhod`(1M), and `ruptime`(1) man pages.

Typical output is illustrated in the following example.

EXAMPLE 28 Output of `ruptime(1)` Program

```
example1 up 9:45, 5 users, load 1.15, 1.39, 1.31
example2 up 2+12:04, 8 users, load 4.67, 5.13, 4.59
example3 up 10:10, 0 users, load 0.27, 0.15, 0.14
example4 up 2+06:28, 9 users, load 1.04, 1.20, 1.65
example5 up 25+09:48, 0 users, load 1.49, 1.43, 1.41
example6 5+00:05, 0 users, load 1.51, 1.54, 1.56
example7 down 0:24
example8 down 17:04
example9 down 16:09
example10 up 2+15:57, 3 users, load 1.52, 1.81, 1.86
```

Status information is periodically broadcast by the `rwho` server processes on each host. The server process also receives the status information. The server also updates a database. This database is interpreted for the status of each host. Servers operate autonomously, coupled only by the local network and its broadcast capabilities.

Use of broadcast is fairly inefficient because broadcast generates a lot of net traffic. Unless the service is used widely and frequently, the expense of periodic broadcasts outweighs the simplicity.

The following example shows a simplified version of the `rwho` server. The sample code receives status information broadcast by other hosts on the network and supplies the status of the host on which the sample code is running. The first task is done in the main loop of the program: Packets received at the `rwho` port are checked to be sure they were sent by another `rwho` server process and are stamped with the arrival time. The packets then update a file with the status of the host. When a host has not been heard from for an extended time, the database routines assume the host is down and logs this information. Because a server might be down while a host is up, this application is prone to error.

EXAMPLE 29 `rwho`(1) Server

```
main()
{
    ...
    sp = getservbyname("who", "udp");
    net = getnetbyname("localnet");
    sin.sin6_addr = inet_makeaddr(net->n_net, in6addr_any);
    sin.sin6_port = sp->s_port;
    ...
    s = socket(AF_INET6, SOCK_DGRAM, 0);
    ...
    on = 1;
    if (setsockopt(s, SOL_SOCKET, SO_BROADCAST, &on, sizeof on)
            == -1) {
        syslog(LOG_ERR, "setsockopt SO_BROADCAST: %m");
        exit(1);
    }
    bind(s, (struct sockaddr *) &sin, sizeof sin);
    ...
    signal(SIGALRM, onalrm);
    onalrm();
    while(1) {
        struct whod wd;
        int cc, whod, len = sizeof from;
        cc = recvfrom(s, (char *) &wd, sizeof(struct whod), 0,
            (struct sockaddr *) &from, &len);
        if (cc <= 0) {
            if (cc == -1 && errno != EINTR)
                syslog(LOG_ERR, "rwhod: recv: %m");
            continue;
        }
        if (from.sin6_port != sp->s_port) {
            syslog(LOG_ERR, "rwhod: %d: bad from port",
                ntohs(from.sin6_port));
            continue;
        }
        ...
        if (!verify( wd.wd_hostname)) {
            syslog(LOG_ERR, "rwhod: bad host name from %x",
                ntohl(from.sin6_addr.s6_addr));
            continue;
        }
        (void) sn

printf(path, sizeof(PATH),

 "%s/whod.%s", RWHODIR, wd.wd_hostname);
        whod = open(path, O_WRONLY|O_CREAT|O_TRUNC

|O_NOFOLLOW, 0666);
        ...
        (void) time(&wd.wd_recvtime);
        (void) write(whod, (char *) &wd, cc);
        (void) close(whod);
    }
    exit(0);
}
```

The second server task is to supply the status of its host. This requires periodically acquiring system status information, packaging it in a message, and broadcasting it on the local network for other `rwho` servers to hear. This task is run by a timer. This task is triggered by a signal.

Status information is broadcast on the local network. For networks that do not support broadcast, use multicast.

Advanced Socket Topics

For most programmers, the mechanisms already described are enough to build distributed applications. This section describes additional features.

Out-of-Band Data

The stream socket abstraction includes out-of-band data. Out-of-band data is a logically independent transmission channel between a pair of connected stream sockets. Out-of-band data is delivered independent of normal data. The out-of-band data facilities must support the reliable delivery of at least one out-of-band message at a time. This message can contain at least one byte of data. At least one message can be pending delivery at any time.

With in-band signaling, urgent data is delivered in sequence with normal data, and the message is extracted from the normal data stream. The extracted message is stored separately. Users can choose between receiving the urgent data in order and receiving the data out of sequence, without having to buffer the intervening data.

Using `MSG_PEEK`, you can peek at out-of-band data. If the socket has a process group, a `SIGURG` signal is generated when the protocol is notified of its existence. A process can set the process group or process ID to deliver `SIGURG` to with the appropriate `fcntl` call, as described in "Interrupt-Driven Socket I/O" on page 157 for `SIGIO`. If multiple sockets have out-of-band data waiting for delivery, a `select` call for exceptional conditions can determine which sockets have such data pending. For more information, see the `fcntl`(2) and `select`(3C) man pages.

A logical mark is placed in the data stream at the point at which the out-of-band data was sent. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal is received, all data up to the mark in the data stream is discarded.

To send an out-of-band message, apply the `MSG_OOB` flag to `send` or `sendto`. To receive out-of-band data, specify `MSG_OOB` to `recvfrom` or `recv`. If out-of-band data is taken in line the `MSG_OOB` flag is not needed. For more information, see the `send`(3C), `sendto`(3C), `recvfrom`(3C), and `recv`(3C) man pages.

The `SIOCATMARK` `ioctl`(2) indicates whether the read pointer currently points at the mark in the data stream.

```
int yes;
ioctl(s, SIOCATMARK, &yes);
```

If *yes* is `1` on return, the next read returns data after the mark. Otherwise, assuming out-of-band data has arrived, the next read provides data sent by the client before sending the out-of-band signal. The routine in the remote login process that flushes output on receipt of an interrupt or quit signal is shown in the following example. This code reads the normal data up to the mark to discard the normal data, then reads the out-of-band byte.

A process can also read or peek at the out-of-band data without first reading up to the mark. Accessing this data when the underlying protocol delivers the urgent data in-band with the normal data, and sends notification of its presence only ahead of time, is more difficult. An example of this type of protocol is TCP, the protocol used to provide socket streams in the Internet family. With such protocols, the out-of-band byte might not yet have arrived when `recv` is called with the `MSG_OOB` flag. In that case, the call returns the error of `EWOULDBLOCK`. Also, the amount of in-band data in the input buffer might cause normal flow control to prevent the peer from sending the urgent data until the buffer is cleared. The process must then read enough of the queued data to clear the input buffer before the peer can send the urgent data.

EXAMPLE 30 Flushing Terminal I/O on Receipt of Out-of-Band Data

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
    int out = FWRITE;
    char waste[BUFSIZ];
    int mark = 0;

    /* flush local terminal output */
    ioctl(1, TIOCFLUSH, (char *) &out);
    while(1) {
        if (ioctl(rem, SIOCATMARK, &mark) == -1) {
            perror("ioctl");
            break;
        }
        if (mark)
            break;
        (void) read(rem, waste, sizeof waste);
    }
    if (recv(rem, &mark, 1, MSG_OOB) == -1) {
        perror("recv");
        ...
    }
    ...
}
```

A facility to retain the position of urgent in-line data in the socket stream is available as a socket-level option, `SO_OOBINLINE`. For more information, see the `getsockopt`(3C) man page. With this socket-level option, the position of urgent data remains. However, the urgent data immediately following the mark in the normal data stream is returned without the `MSG_OOB` flag. Reception of multiple urgent indications moves the mark, but does not lose any out-of-band data.

Nonblocking Sockets

Some applications require sockets that do not block. For example, a server would return an error code, not executing a request that cannot complete immediately. This error could cause the process to be suspended, awaiting completion. After creating and connecting a socket, issuing a fcntl(2) call, as shown in the following example, makes the socket nonblocking.

EXAMPLE 31 Set Nonblocking Socket

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL, 0) == -1) {
    perror("fcntl F_GETFL");
    exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY) == -1) {
    perror("fcntl F_SETFL, FNDELAY");
    exit(1);
}
```

When performing I/O on a nonblocking socket, check for the error `EWOULDBLOCK` in `errno.h`, which occurs when an operation would normally block. `accept`, `connect`, `send`, `recv`, `read`, and `write` can all return `EWOULDBLOCK`. If an operation such as a `send` cannot be done in its entirety but partial writes work, as when using a stream socket, all available data is processed. The return value is the amount of data actually sent. For more information, see the accept(3C), connect(3C), send(3C), read(2), write(2), and recv(3C) man pages.

Asynchronous Socket I/O

Asynchronous communication between processes is required in applications that simultaneously handle multiple requests. Asynchronous sockets must be of the `SOCK_STREAM` type. To make a socket asynchronous, you issue a fcntl(2) call, as shown in the following example.

EXAMPLE 32 Making a Socket Asynchronous

```
#include <fcntl.h>
#include <sys/file.h>
...
int fileflags;
int s;
...
s = socket(AF_INET6, SOCK_STREAM, 0);
...
if (fileflags = fcntl(s, F_GETFL ) == -1) {
    perror("fcntl F_GETFL");
    exit(1);
}
if (fcntl(s, F_SETFL, fileflags | FNDELAY | FASYNC) == -1) {
    perror("fcntl F_SETFL, FNDELAY | FASYNC");
    exit(1);
}
```

After sockets are initialized, connected, and configured as nonblocking and asynchronous, communication is similar to reading and writing a file asynchronously. Initiate a data transfer by using `send`, `write`, `recv`, or `read`. For more information, see the `send`(3C), `write`(2), `recv`(3C), and `read`(2) man pages. A signal-driven I/O routine completes a data transfer, as described in the next section.

Interrupt-Driven Socket I/O

The `SIGIO` signal notifies a process when a socket, or any file descriptor, has finished a data transfer. The steps in using `SIGIO` are as follows:

1. Set up a `SIGIO` signal handler with the `signal` or `sigvec` calls. For more information, see the `signal`(3C) man page.
2. Use `fcntl` to set the process ID or process group ID to route the signal to its own process ID or process group ID. The default process group of a socket is group `0`. For more information, see the `fcntl`(2) man page.
3. Convert the socket to asynchronous, as shown in "Asynchronous Socket I/O" on page 156.

The following sample code enables receipt of information on pending requests as the requests occur for a socket by a given process. With the addition of a handler for `SIGURG`, this code can also be used to prepare for receipt of `SIGURG` signals.

EXAMPLE 33 Asynchronous Notification of I/O Requests

```
#include <fcntl.h>
#include <sys/file.h>
 ...
signal(SIGIO, io_handler);
/* Set the process receiving SIGIO/SIGURG signals to us. */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
    perror("fcntl F_SETOWN");
    exit(1);
}
```

Signals and Process Group ID

For `SIGURG` and `SIGIO`, each socket has a process number and a process group ID. These values are initialized to zero, but can be redefined at a later time with the `F_SETOWN` `fcntl` command, as in the previous example. A positive third argument to `fcntl` sets the socket's process ID. A negative third argument to `fcntl` sets the socket's process group ID. The only allowed recipient of `SIGURG` and `SIGIO` signals is the calling process. A similar `fcntl`, `F_GETOWN`, returns the process number of a socket. For more information, see the `fcntl`(2) man page.

You can also enable reception of `SIGURG` and `SIGIO` by using `ioctl` to assign the socket to the user's process group. For more information, see the `ioctl`(2) man page.

```
/* oobdata is the out-of-band data handling routine */
sigset(SIGURG, oobdata);
int pid = -getpid();
if (ioctl(client, SIOCSPGRP, (char *) &pid) < 0) {
    perror("ioctl: SIOCSPGRP");
}
```

Selecting Specific Protocols

If the third argument of the `socket` call is `0`, `socket` selects a default protocol to use with the returned socket of the type requested. The default protocol is usually correct, and alternate choices are not usually available. When using raw sockets to communicate directly with lower-level protocols or lower-level hardware interfaces, set up de-multiplexing with the protocol argument. For more information, see the `socket`(3C) man page.

Using raw sockets in the Internet family to implement a new protocol on IP ensures that the socket only receives packets for the specified protocol. To obtain a particular protocol, determine the protocol number as defined in the protocol family. For the Internet family, use one of the library routines that are discussed in "Standard Routines" on page 143, such as `getprotobyname`.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
...
pp = getprotobyname("newtcp");
s = socket(AF_INET6, SOCK_STREAM, pp->p_proto);
```

Using `getprotobyname` results in a socket `s` by using a stream-based connection, but with a protocol type of `newtcp` instead of the default `tcp`.

Address Binding

For addressing, TCP and UDP use a 4-tuple of:

- Local IP address
- Local port number
- Foreign IP address
- Foreign port number

TCP requires these 4-tuples to be unique. UDP does not. User programs do not always know proper values to use for the local address and local port, because a host can reside on multiple networks. The set of allocated port numbers is not directly accessible to a user. To avoid these problems, leave parts of the address unspecified and let the system assign the parts appropriately when needed. Various portions of these tuples can be specified by various parts of the sockets API:

`bind`	Local address or local port or both. For more information, see the `bind`(3C) man page.
`connect`	Foreign address and foreign port. For more information, see the `connect`(3C) man page.

A call to `accept` retrieves connection information from a foreign client. This causes the local address and port to be specified to the system even though the caller of `accept` did not specify anything. The foreign address and foreign port are returned.

A call to `listen` can cause a local port to be chosen. If no explicit `bind` has been done to assign local information, `listen` assigns an ephemeral port number.

A service that resides at a particular port can `bind` to that port. Such a service can leave the local address unspecified if the service does not require local address information. The local address is set to `in6addr_any`, a variable with a constant value in `<netinet/in.h>`. If the local port does not need to be fixed, a call to `listen` causes a port to be chosen. Specifying an address of `in6addr_any` or a port number of 0 is known as *wildcarding*. For `AF_INET`, `INADDR_ANY` is used in place of `in6addr_any`. For more information, see the `bind`(3C) and `listen`(3C) man pages.

The wildcard address simplifies local address binding in the Internet family. The following sample code binds a specific port number that was returned by a call to `getaddrinfo` to a socket and leaves the local address unspecified:

```
#include <sys/types.h>
#include <netinet/in.h>
...
struct addrinfo  *aip;
...
if (bind(sock, aip->ai_addr, aip->ai_addrlen) == -1) {
    perror("bind");
    (void) close(sock);
    return (-1);
}
```

Each network interface on a host typically has a unique IP address. Sockets with wildcard local addresses can receive messages that are directed to the specified port number. Messages that are sent to any of the possible addresses that are assigned to a host are also received by sockets with wildcard local addresses. To allow only hosts on a specific network to connect to the server, a server binds the address of the interface on the appropriate network.

Similarly, a local port number can be left unspecified, in which case the system selects a port number. For example, to bind a specific local address to a socket, but to leave the local port number unspecified, you could use `bind` as follows:

```
bzero (&sin, sizeof (sin));
(void) inet_pton (AF_INET6, "::ffff:127.0.0.1", sin.sin6_addr.s6_addr);
sin.sin6_family = AF_INET6;
sin.sin6_port = htons(0);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

The system uses two criteria to select the local port number:

- Internet port numbers less than 1024 (`IPPORT_RESERVED`) are reserved for privileged users. Nonprivileged users can use any Internet port number that is greater than 1024. The largest Internet port number is 65535.
- The port number is not currently bound to some other socket.

The port number and IP address of the client are found through `accept` or `getpeername`. For more information, see the `accept`(3C) and `getpeername`(3C) man pages.

In certain cases, the algorithm used by the system to select port numbers is unsuitable for an application due to the two-step creation process for associations. For example, the Internet file transfer protocol specifies that data connections must always originate from the same local port. Duplicate associations are avoided by connecting to different foreign ports. In this situation, the system would disallow binding the same local address and local port number to a socket if a previous data connection's socket still existed.

To override the default port selection algorithm, you must perform an option call before address binding:

```
int on = 1;
...
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &on, sizeof on);
bind(s, (struct sockaddr *) &sin, sizeof sin);
```

With this call, local addresses already in use can be bound. This binding does not violate the uniqueness requirement. The system still verifies at connect time that any other sockets with the same local address and local port do not have the same foreign address and foreign port. If the association already exists, the error `EADDRINUSE` is returned.

Socket Options

You can set and get several options on sockets through `setsockopt` and `getsockopt`. For example, you can change the send or receive buffer space. For more information, see the `setsockopt`(3C) and `getsockopt`(3C) man pages.

The general forms of the calls are in the following formats:

```
setsockopt(s, level, optname, optval, optlen);

getsockopt(s, level, optname, optval, optlen);
```

The operating system can adjust the values appropriately at any time.

The arguments of `setsockopt` and `getsockopt` calls are in the following list:

s	Socket on which the option is to be applied
level	Specifies the protocol level, such as socket level, indicated by the symbolic constant `SOL_SOCKET` in `sys/socket.h`

optname	Symbolic constant defined in `sys/socket.h` that specifies the option
optval	Points to the value of the option
optlen	Points to the length of the value of the option

For `getsockopt`, *optlen* is a value-result argument. This argument is initially set to the size of the storage area pointed to by *optval*. On return, the argument's value is set to the length of storage used.

When a program needs to determine an existing socket's type, the program should invoke `inetd` by using the `SO_TYPE` socket option and the `getsockopt` call:

```
#include <sys/types.h>
#include <sys/socket.h>

int type, size;

size = sizeof (int);
if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &type, &size) < 0) {
    ...
}
```

After `getsockopt`, `type` is set to the value of the socket type, as defined in `sys/socket.h`. For a datagram socket, `type` would be `SOCK_DGRAM`.

Socket Level Properties

Starting with the Oracle Solaris 11.2 release, you can use the `SO_FLOW_SLA` option to set the service-level properties for the socket. A socket application using the `SO_FLOW_SLA` socket option causes the system to create a system flow, which is an enforcement mechanism for the service-level properties. You can use `flowadm` to observe the system flows. For more information, see the `flowadm`(1M) man page. These system flows have the prefix <id>`.sys.sock`.

The `pfiles` prints the `SO_FLOW_SLA` socket option with other socket options. For more information, see the `pfiles`(1) man page.

Note - You can set the socket level properties only for `TCP` and `UDP` sockets.

The usage of `SO_FLOW_SLA` socket option is described in the following example.

```
#include <sys/types.h>
#include <sys/socket.h>

extern struct sockaddr *serv_addr;

int fd;
mac_flow_props_t mprop;
mac_flow_props_t mprop_result;

fd = socket(AF_INET, SOCK_STREAM, 0);

mprop.mfp_version = MAC_FLOW_PROP_VERSION1;
mprop.mfp_mask = MFP_MAXBW | MFP_PRIORITY;
mprop.mfp_priority = MFP_PRIO_HIGH;
mprop.mfp_maxbw = 100000000;           /* in bits per second */
setsockopt(fd, SOL_SOCKET, SO_FLOW_SLA, &mprop, sizeof (mprop));

connect(fd, serv_addr, sizeof(*serv_addr));

getsockopt(fd, SOL_SOCKET, SO_FLOW_SLA, &mprop_result, sizeof (mprop_result));
```

In the example, the `TCP` client socket is created along with the system flow. The flow is set to a high priority and the maximum bandwidth is set to 100Mbps.

The system flow is created for the socket by calling `connect()` or `accept()` functions after `setsockopt`. If either `accept()` or `connect()` function is already called, setting `SO_FLOW_SLA` will create a flow. Properties of the flow are set according to the values specified in `mac_flow_props_t` structure. This structure is passed as a pointer to `setsockopt` as an `optval` argument. You can know the status of the flow creation by using `getsockopt`. The status is stored in the `mprop_result.mfp_status` field. The `mac_flow_props_t` structure is defined as follows.

```
typedef struct mac_flow_props_s {
int  mfp_version;
uint32_t   mfp_mask;
int  mfp_priority;   /* flow priority */
uint64_t   mfp_maxbw;   /* bandwidth limit in bps */
int  mfp_status;  /* flow create status for getsockopt */
} mac_flow_props_t;
```

The following list describes the fields of the `mac_flow_props_t` structure.

`mfp_version`	Denotes the version of the `mac_flow_props_t` structure. Currently, `mfp_version` can only be set to 1.

```
#define MAC_FLOW_PROP_VERSION1
```

`mfp_mask` Denotes the bit mask values. The following bit mask values are valid.

- `MRP_MAXBW`
- `MRP_PRIORITY`

`mfp_priority` Denotes the priority of processing the packets that belong to the socket. The following priority values are valid.

- `MFP_PRIO_NORMAL`
- `MFP_PRIO_HIGH`

`mfp_maxbw` Denotes the maximum bandwidth allotted to the socket in bits per second. Value of 0 means all the packets of socket must be dropped.

`mfp_status` Denotes the status of the flow creation. You can obtain the status of flow creation by calling `getsockopt`. `getsockopt` sets the `mfp_status` field. A value of 0 means a flow is successfully created. In case of an error, this field is set to one of the following error codes.

- `EPERM`: No Privilege.
- `ENOTCONN`: If the call is made before the application does a connect or bind.
- `EOPNOTSUPP`: Flow creation is not supported for this socket.
- `EALREADY`: Flow with identical attributes exists.
- `EINPROGRESS`: Flow is being created.

`inetd` Daemon

The `inetd` daemon is invoked at startup time and is now configured by using `smf`. The configuration was previously performed by `/etc/inet/inetd.conf` file. For more information, see the `inetd`(1M), and `smf`(7) man pages.

Use `inetconv` to convert the configuration file content into SMF format services, and then manage these services using `inetadm` and `svcadm`. For more information, see the `inetconv`(1M), `inetadm`(1M) and `svcadm`(1M) man pages.

The `inetd` daemon polls each socket, waiting for a connection request to the service corresponding to that socket. For `SOCK_STREAM` type sockets, `inetd` accepts using `accept` on the listening socket, forks using `fork`, duplicates using `dup` the new socket to file descriptors `0` and `1` (`stdin` and `stdout`), closes other open file descriptors, and executes using `exec` the appropriate server. For more information, see the `accept`(3C), `fork`(2), `dup`(2), and `exec`(2) man page.

The primary benefit of using `inetd` is that services not in use do not consume system resources. A secondary benefit is that `inetd` does most of the work to establish a connection. The server started by `inetd` has the socket connected to its client on file descriptors `0` and `1`. The server can immediately read, write, send, or receive. Servers can use buffered I/O as provided by the `stdio` conventions, as long as the servers use `fflush` when appropriate. For more information, see the `fflush`(3C) man page.

The `getpeername` routine returns the address of the peer (process) connected to a socket. This routine is useful in servers started by `inetd`. For example, you could use this routine to log the Internet address such as `fec0::56:a00:20ff:fe7d:3dd2`, which is conventional for representing the IPv6 address of a client. An `inetd` server could use the following sample code:

```
struct sockaddr_storage name;
int namelen = sizeof (name);
char abuf[INET6_ADDRSTRLEN];
struct in6_addr addr6;
struct in_addr addr;

if (getpeername(fd, (struct sockaddr *) &name, &namelen) == -1) {
    perror("getpeername");
    exit(1);
} else {
    addr = ((struct sockaddr_in *)&name)->sin_addr;
    addr6 = ((struct sockaddr_in6 *)&name)->sin6_addr;
    if (name.ss_family == AF_INET) {
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6 &&
               IN6_IS_ADDR_V4MAPPED(&addr6)) {
            /* this is a IPv4-mapped IPv6 address */
            IN6_MAPPED_TO_IN(&addr6, &addr);
            (void) inet_ntop(AF_INET, &addr, abuf, sizeof (abuf));
    } else if (name.ss_family == AF_INET6) {
            (void) inet_ntop(AF_INET6, &addr6, abuf, sizeof (abuf));

    }
    syslog("Connection from %s\n", abuf);
}
```